Integration of Multiple Constraints in ACG

Jiří Maršík[1,2,3] and Maxime Amblard[1,2,3]

[1] Université de Lorraine, Laboratoire lorrain de recherche en informatique et ses applications, UMR 7503, Vandoeuvre-lès-Nancy, 54500, France
[2] INRIA, Villers-lès-Nancy, 54600, France
[3] CNRS, Loria, Vandoeuvre-lès-Nancy, UMR 7503, 54500, France
{jiri.marsik, maxime.amblard}@loria.fr

Abstract. This proposal is a first step towards a wide-coverage Abstract Categorial Grammar (ACG) that could be used to automatically build discourse-level representations. We focus on the challenge of integrating the treatment of disparate linguistic constraints in a single ACG and propose a generalization of the formalism: Graphical Abstract Categorial Grammars.

Keywords: abstract categorial grammars, grammar engineering, grammatical formalisms, formal grammars, computational linguistics

Abstract Categorial Grammars (ACGs) [1] have shown to be a viable formalism for elegantly encoding the dynamic nature of discourse. Proposals based on continuation semantics [2] have tackled topics such as event anaphora [3], SDRT discourse structure [4] and modal accessibility constraints [5]. However, all of these treatments only consider tiny fragments of languages. We are interested in building a wide-coverage grammar which integrates and reconciles the existing formal treatments of discourse and allows us to study their interactions and to build discourse representations automatically.

A presupposed condition to actually using an ACG to describe discourse in a large scope is to have a large scale ACG grammar in the first place. The work presented here was motivated by this quest for a wide-coverage ACG grammar, as seen from a point of view of language as a system of largely orthogonal constraints. Encoding ancillary constraints in a type system can make the system hard to read and understand. If multiple constraints written in the style of [6] are to be enforced in the same grammar, we advocate for extending the formalism to prevent the incidental complexity that would otherwise emerge.

We begin with a short review of Abstract Categorial Grammars where we highlight the ways in which Abstract Categorial Grammars facilitate decomposition of linguistic description. We follow that by a closer look at an example of a linguistic constraint. This discussion motivates our proposed extension. We then present the Graphical Abstract Categorial Grammars and consider some of its formal properties. We then finish with an illustration of our approach which incorporates linguistic contributions from multiple sources into a single coherent and simple grammar and examine the limitations and challenges of adopting this approach.

1 Abstract Categorial Grammars

We first review the grammatical framework in which we conduct our work. Abstract Categorial Grammars are built upon two mathematical structures: *(higher-order) signatures* and *lexicons*.

1.1 Higher-Order Signatures

A **higher-order signature** is a set of elements that we call *constants*, each of which is associated with a type. Formally, it is defined as a triple $\Sigma = <A, C, \tau>$, where:

- C is the (finite) set of constants
- A is a (finite) set of atomic types
- τ is the type-associating mapping from C to $\mathcal{T}(A)$, the set of types built over A

In our case, $\mathcal{T}(A)$ is the implicative fragment of linear and intuitionistic logic with A being the atomic propositions. This means that $\mathcal{T}(A)$ contains all the $a \in A$ and all the $\alpha \multimap \beta$ and $\alpha \to \beta$ for $\alpha, \beta \in \mathcal{T}(A)$.

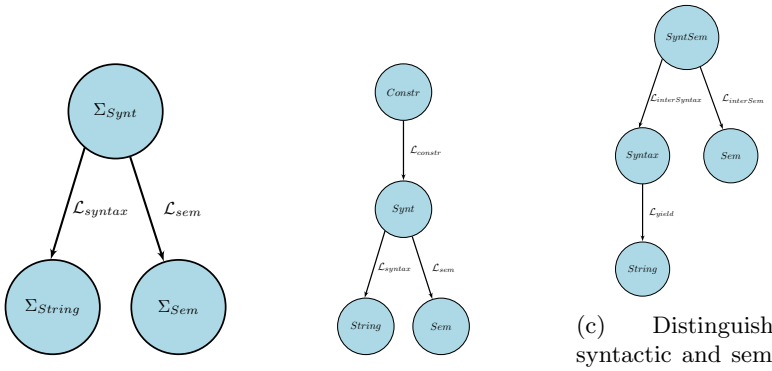
A signature $\Sigma = <A, C, \tau>$, by itself, already lets us define an interesting set of structures, that is the set $\Lambda(\Sigma)$ of *well-typed lambda terms* built upon the signature Σ. To make this structure even more useful, we often focus ourselves only on terms that have a specific *distinguished type*. Using this notion of a signature of typed constants and some distinguished type, we can describe languages of, e.g. tree-like (and by extension string-like), lambda terms.

1.2 Lexicons

The idea of a signature is coupled with the one of **lexicon**, which is a mapping between two different signatures (mapping the constants of one into well-typed terms of the other). Formally speaking, a lexicon $\mathcal{L}$ from a signature $\Sigma_1 = <A_1, C_1, \tau_1>$ (which we call the abstract signature) to a signature $\Sigma_2 = <A_2, C_2, \tau_2>$ (which we call the object signature) is a pair $<F, G>$ such that:

- G is a mapping from C_1 to $\Lambda(\Sigma_2)$ assigning to every constant of the abstract signature a term in the object signature, which can be understood as its interpretation/implementation/realization.
- F is a mapping from A_1 to $\mathcal{T}(A_2)$ which links the abstract-level types with the object-level types that they are realized as.
- F and G are compatible, meaning that for any $c \in C_1$, we have $\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c))$ (we will be using $\hat{F}$ and $\hat{G}$ to refer to the homomorphic extensions of F and G to $\mathcal{T}(A_1)$ and $\Lambda(\Sigma_1)$ respectively).

Now we have enough machinery in hand to describe how ACGs define languages. Given some signature and a distinguished type, we can talk of the *abstract language*, which is the set of well-typed terms built on the signature and



(a) Connecting form with meaning.

(b) Adding a constraint (Constraint).

(c) Distinguishing syntactic and semantic ambiguities (Elaboration).

Fig. 1: Diagrams of systems of ACGs.

having the distinguished type. If we then also consider a lexicon and its abstract signature, then the language we get by mapping the abstract language of the signature through the lexicon is called an *object language*.

An ACG for us will then be just a collection of signatures with their distinguished types and the lexicons connecting these signatures, which we will write down in diagrams such as the ones in Figure 1.

The most common pattern will have us using two object signatures, Σ_{String} and Σ_{Sem}, for the surface forms of utterances (strings) and their logical forms (logical propositions) and an abstract signature, Σ_{Synt}, which is connected to both of the object signatures via lexicons (as we can see in Figure 1(a)). Parsing is then just a matter of inverting the surface lexicon to get the abstract term and then applying to it the logical lexicon. Generation is symmetric, we simply invert the logical lexicon and apply the surface lexicon.

1.3 Systems of ACGs

Besides the pattern in Figure 1(a), there have been other techniques of decomposing linguistic description in the ACG literature. Here we will focus on two that are relevant to our work.

The first constitutes the Constraint pattern used throughout [6]. Imagine we want our grammar to enforce a particular linguistic constraint, say that tensed clauses form islands for quantifier scope, and suppose we want to proceed by refining our types so as to make linguistic structures violating the constraint untypable. We could dive into our grammar and rewrite it to use the new types. However, we can also keep the existing grammar intact and complement it with a signature and a lexicon that translates that signature to the abstract-most signature of our existing grammar (Figure 1(b)). Usually, the new signature will resemble the prior abstract-most one, albeit with more fine-grained types necessary to express the constraint in question.

This pattern is practical for examining linguistic constraints one-by-one and demonstrating that the formalism is capable of expressing them individually. However, this pattern does not serve us much in the context of ACGs when we try to build a grammar which incorporates many of these at the same time. This is due to the fact that after grafting on the first constraint, its added signature now becomes the abstract-most signature in the grammar. Any other constraint that would be added using the Constraint pattern would have to be translated into this new signature. This would mean that the type systems of newly added constraints would have to re-enforce all of the constraints introduced before, therefore nullifying almost any benefit of the pattern.

Another pattern we will briefly mention is the Elaboration pattern in which an edge in the diagram (a lexicon) is expanded into two edges and an intermediate node (signature). We can illustrate this pattern on Tree Adjoining Grammars [7]. Suppose we have a signature describing TAG derivation trees and a lexicon which maps them to their yield. We might want to make this grammar richer, and arguably more synoptic, by elaborating the translation from derivation trees to yields by introducing derived trees and describe separately how both derivation trees produce derived trees and how derived trees produce the yields. This way, we can make emerge a potentially useful linguistic structure.

This pattern was used in [8] to elaborate the translation from the common Montague-style [9] higher-order signature to its yield by introducing lower-order syntactic structures more reminiscent of TAG or CFG trees. This helps make a clear distinction between ambiguities caused by syntax and by semantics by segregating the purely syntactic description in the $Syntax$ signature from the issues of operator scope that have to be solved in the $SyntSem$ signature (Figure 1(c)). However, in classical ACGs, we cannot exploit this pattern as much as we would like. If we were to use this grammar as a kernel on top of which we would like to express a syntactic constraint, we would not be able to attach the constraint's signature directly to the $Syntax$ node but we would instead need to attach it to $SyntSem$[4]. This means that the syntactic constraint would end up being expressed in terms of the types of the syntax-semantics interface ($SyntSem$) rather than just pure syntax ($Syntax$), which only adds unnecessary complexity.

2 The Problem of Multiple Constraints

We will consider several linguistic constraints that have been given formal treatments in grammatical formalisms.

In French, negation is signalled by prepending the particle *ne* to the negated verb in conjunction with using a properly placed accompanying word, such as a negative determiner, in one of the verb's arguments. This phenomenon has been elegantly formalized in the Frigram interaction grammar [10].

[4] This is due to the fact that ACG diagrams are always arborescences, in which the root node represents the abstract language from whose terms are generated the terms of all the object languages.

(1) Jean **ne** parle à **aucun** collègue.
(Jean speaks to no colleague.)

(2) Jean **ne** parle à la femme d'**aucun** collègue.
(Jean speaks to the wife of no colleague.)

(3) **Aucun** collègue de Jean **ne** parle à sa femme.
(No colleague of Jean's speaks to his wife.)

We see here that the negative determiner *aucun* can be present in the subject or the object of the negated verb and it can modify the argument directly or be attached to one of its complements. Furthermore, we note that omitting either the word *ne* or the word *aucun* while keeping the other produces a sentence which is considered ungrammatical.

This difference in syntactic behavior between noun phrases that contain a negative determiner and those that do not has implications for our grammar. Since two terms that have an identical type in an ACG signature can be freely interchanged in any sentence, we are forced to assign two different types to these two different kinds of noun phrases.

This leads us to a grammar in which we subdivide the atomic types N and NP into subtypes that reflect whether or not they contain a negative determiner inside. Types of the other constants, such as the preposition *de* seen in (2) and (3), will constrain their arguments to have compatible features on their types and will propagate the information carried in the features to its result type, e.g.:

$$N_{de_1} : NP_{NEG=F} \multimap N_{NEG=F} \multimap N_{NEG=F}$$
$$N_{de_2} : NP_{NEG=F} \multimap N_{NEG=T} \multimap N_{NEG=T}$$
$$N_{de_3} : NP_{NEG=T} \multimap N_{NEG=F} \multimap N_{NEG=T}$$

In the above, we elaborate the types NP and N with features ($NEG{=}F$ and $NEG{=}T$) and we give the different types for the preposition *de* in the fragment for negation (N). The types accept an NP and an N as arguments with any combination of values for the feature NEG, except for the case when both the prepositional NP and the N being modified both contain free negative determiners (i.e. there is no $N_{de_4} : NP_{NEG=T} \multimap N_{NEG=T} \multimap N_{NEG=} \ldots$). This encodes a constraint that there can be only one free negative determiner per phrase (free as in not hidden inside an embedded clause). Besides this constraint, the types also dictate how the information about negative determiners should propagate from the argument constituent to the complex constituent (in this case, it is simple disjunction).

Enforcing other constraints leads us to subdividing our "atomic" types even further (e.g. the authors of [6] add features to the S and NP types to implement constraints about extraction). Other phenomena, such as agreement on morphological properties like number, gender, person or case, intuitively lead us to make our types even more specific.

If we were to use this approach to write a grammar that enforces multiple constraints at the same time, we would end up with complicated types, like the one below, which provide complete specifications of the various possible situations (in this grammar (C), the preposition *de* has 12 different types).

$C_{de_{11}} : NP_{NEG=T,VAR=F,NUM=PL} \multimap N_{NEG=F,NUM=SG} \multimap N_{NEG=T,NUM=SG}$

This creates two problems. Firstly, the number of such types grows exponentially with the number of features added. This can be fixed by introducing dependent types into the type system as in [11]. However, while this allows us to abstract over the combinations of feature values and write our grammar down concisely, it does not take away the complexity. The treatments of the various linguistic phenomena are all expressed in the same types making it hard to see whether they are independent or not. Since the treatments cannot be considered in isolation, reasoning about the entire grammar becomes difficult and so does enhancing it with more constraints. This is a fatal problem for a grammar which strives to cover a wide range of linguistic facts. We firmly believe that simplicity is a fundamental requirement for constructing a large and robust grammar and our proposal aims to reclaim that simplicity.

In our grammar, we would like to combine several constraints (Figure 1(b)), and possibly to also separate the syntactic ambiguities from the purely semantic ones (Figure 1(c)). However, trying to mix these strategies in the ACG framework forces us to solve all the constraints in a single type signature or contaminate the syntax-semantics interface with the implementation details of the syntactic layer, both of which introduce incidental complexity we want to avoid.

We would like to have a system which would be characterized by a diagram like the one on Figure 2 where the constraint signatures delimit the legal syntactic structures independently of each other and without interfering with the syntax-semantics interface. However, ACG diagrams are limited to arborescences and we are obliged to generalize them in order to get the expected interpretation.

3 Graphical ACGs

We define a *graphical abstract categorial grammar* as a directed acyclic graph (DAG) whose nodes are labeled with signatures (and distinguished types) and whose edges are labeled with lexicons, in other words, a mathematical reification of an ACG diagram that has been generalized from arborescences to DAGs. We then search for an appropriate semantics for these structures, i.e. how to determine what languages are defined by a graphical ACG.

3.1 Nodes as Languages

We first follow a paradigm in which nodes of the diagrams are interpreted as languages with the edges telling us how these languages are defined in terms of each other. A single arrow leading to a language means that the target language is produced from the source by mapping it through a lexicon. We now argue that the suitable meaning of two or more edges arriving at the same node is intersection of languages based both on the simplicity of the resulting definitions and on our expectations about the desired semantics.

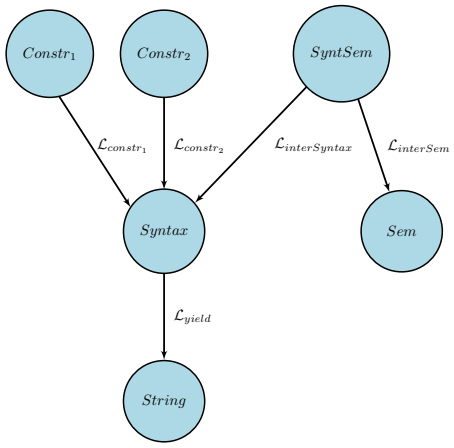


Fig. 2: A graphical ACG that implements two independent syntactic constraints and distinguishes syntactic and semantic ambiguities.

In an ACG diagram, a node with no inbound edges stands for an abstract language. This language is defined as the set of terms having the correct type. If a node has an inbound edge, and therefore a parent, then the elements of its language are also obliged to have an antecedent in the parent language. It is a small step to go from this definition to the following: the language of a node is the set of terms having the correct type and an antecedent in the language of any of its parent nodes. This correctly characterizes the current use of ACG diagrams, recognizing abstract languages as a special case of object languages. Furthermore, this restatement generalizes to DAGs and gives us the desired semantics for implementing multiple constraints: intersection.

We can formalize the above definitions by introducing the notions of *intrinsic* and *extrinsic* languages associated with some node v in a graphical ACG $\mathcal{G}$:

$$\mathcal{I}_{\mathcal{G}}(v) = \{t \in \Lambda(\Sigma_v) \mid \vdash_{\Sigma_v} t : S_v\}$$

$$\mathcal{E}_{\mathcal{G}}(v) = \mathcal{I}_{\mathcal{G}}(v) \cap \bigcap_{(u,v)\in E} \mathcal{L}_{(u,v)}(\mathcal{E}_{\mathcal{G}}(u))$$

The intrinsic language is just the set of terms built on the node's signature and having the node's distinguished type. The extrinsic language is established by taking the extrinsic languages of its predecessors, mapping them through lexicons and taking their intersection, or just taking the node's intrinsic language if it has no predecessors.

We then examine the relationship between the languages defined by ACGs and graphical ACGs (G-ACGs). Intrinsic languages correspond exactly to abstract languages and therefore the sets of languages definable by both are equal.

$$\mathcal{I} = \mathcal{A}$$

G-ACG extrinsic languages correspond to ACG object languages with intersection. More formally, while ACG object languages are ACG abstract languages closed on transformation by a lexicon, G-ACG extrinsic languages are ACG abstract languages closed on transformation by a lexicon and intersection[5].

$$\mathcal{O} = \mathcal{A}^{\mathcal{L}}$$
$$\mathcal{E} = \mathcal{A}^{\mathcal{L}\cap}$$

If we want our grammatical framework to be modular w.r.t. the different linguistic constraints it encodes, we essentially need intersection[6]. Not only would we want a framework in which intersection is possible, we would also like it to be an easy operation. Graphical ACGs make intersections trivial to encode and as for expressive power, we know, from the equations above, that object languages are as expressive as extrinsic languages iff object languages are closed on intersection (which is, at this moment, conjectured to be false). This means that graphical ACGs are either a conservative extension providing a more convenient way of expressing intersection or are extending ACGs by adding only intersection, which enables constraint-based composition of grammars.

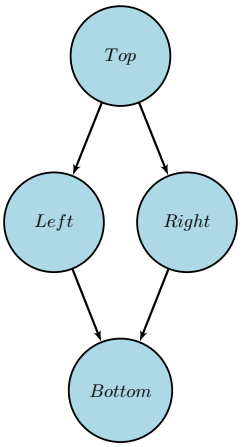


Fig. 3: The diamond-shaped G-ACG $\mathcal{D}$.

While the interpretation of G-ACGs given above does have some nice properties, it fails to predict the intuitive understanding of more complex ACG diagrams. This is most visible in the diamond-shaped grammar on Figure 3. If we take a term from $\mathcal{E}_{\mathcal{D}}(Bottom)$, we know it has antecedents both in $\mathcal{E}_{\mathcal{D}}(Left)$ and $\mathcal{E}_{\mathcal{D}}(Right)$. However, these two do not have to share an antecedent in $\mathcal{E}_{\mathcal{D}}(Top)$.

[5] Can be shown by induction on the topological ordering of any given graphical ACG.
[6] If I have a grammar that enforces A and a grammar that enforces B, then I want to have access to the grammar of the language where both A and B are enforced.

This contradicts the generative story one might imagine in which a single term from Top generates terms in $Left$ and $Right$ which generate a term in $Bottom$.

We can observe another peculiarity on a more practical example. Consider the G-ACG in Figure 2. In classical ACGs, one can always take an element in a signature, like Sem, and by transduction find its corresponding element in an another signature, like $Syntax$. However, in a G-ACG such as this, it is possible that the $Syntax$ term we obtain by transduction does not belong to $\mathcal{E}_{\mathcal{G}}(Syntax)$ because it lacks antecedents in either $Constr_1$ or $Constr_2$. This means that the notion of an extrinsic language is not capable to give us the set of all meaning terms in Sem which actually correspond to syntactically correct sentences in this G-ACG.

The above characteristics motivated us to explore alternative interpretations of G-ACGs. We will present one such alternative now, which exchanges the language-algebraic point of view for a generative one.

3.2 Nodes as Terms

In the new paradigm, we interpret the nodes of the graph as terms and the edges as statements that one term is mapped into another using a lexicon. This leads us to the definition of the *pangraphical*[7] language of a node u in a G-ACG $\mathcal{G}$.

A term t belongs to $\mathcal{P}_{\mathcal{G}}(u)$ whenever there exists a labeling T of the nodes of the graph such that:

- $T_u = t$.
- For all $v \in V(G)$, $\vdash_{\Sigma_v} T_v : S_v$.
- For all $(v, w) \in E(G)$, $\mathcal{L}_{(v,w)}(T_v) = T_w$.

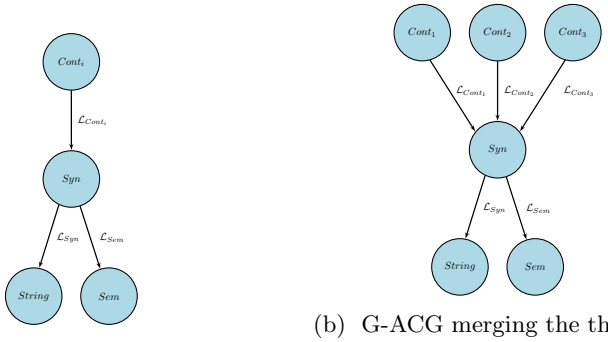
If we compare this new interpretation of G-ACGs to the former one, we find out that in the case when the graph of the grammar is an arborescence, they are actually equivalent. This means that in classical ACGs, where all the diagrams are arborescences, the two metaphors (nodes as languages and nodes as terms) can be, and are, used interchangeably. However, as we start to consider non-arborescent graphs, we find, interestingly, that the two paradigms diverge (i.e. $\exists \mathcal{G}, u.\ \mathcal{E}_{\mathcal{G}}(u) \neq \mathcal{P}_{\mathcal{G}}(u)$).

The newly defined pangraphical languages solve the problem of extrinsic languages giving us counter-intuitive interpretations for some specific G-ACGs. Specifically, the members of $\mathcal{P}_{\mathcal{D}}(Bottom)$ in the diamond grammar have a single antecedent in $\mathcal{P}_{\mathcal{D}}(Top)$ and the language $\mathcal{P}_{\mathcal{G}}(Sem)$ (from Figure 2) contains only meanings expressible by syntactically correct sentences.

Pangraphical languages turn out to be at least as expressive as the extrinsic languages[8]. The proof is carried out by transforming a G-ACG such that

[7] As opposed to extrinsic languages, which are constrained only by their predecessors in the graph, pangraphical languages are constrained by the entire graph.

[8] This might come as no surprise given the extended domain of constraints compared to extrinsic languages (constrained by the entire graph as opposed to just the node's predecessors).



(a) An extraction constraint

(b) G-ACG merging the three extraction constraints.

Fig. 4: Combining the constraints on extraction.

a particular node will have the same pangraphical language in the latter G-ACG as the extrinsic language it had in the former. The nodes of the newly constructed G-ACG correspond to paths in the former one terminating in the node in question[9]. Thus the labelling of nodes with terms which certifies a given term's presence in the pangraphical language of the new G-ACG also serves as a proof of its presence in the extrinsic language of the old G-ACG and vice versa.

This gives us the following ladder of expressivity

$$\mathcal{I} \subseteq \mathcal{E} \subseteq \mathcal{P}$$

which can be complemented with the following series

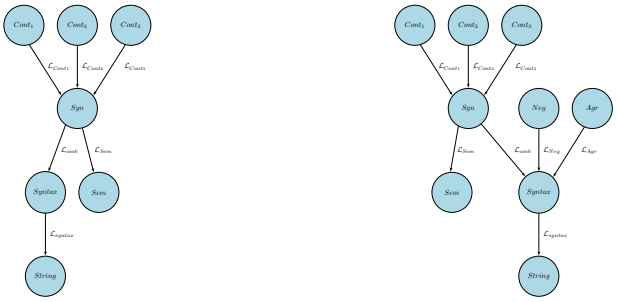
$$\mathcal{I}_{\mathcal{G}}(u) \supseteq \mathcal{E}_{\mathcal{G}}(u) \supseteq \mathcal{P}_{\mathcal{G}}(u)$$

4 Illustration

In this final section, we assemble a G-ACG which integrates the French negation constraint discussed in Section 2, the constraints on extraction introduced in [6] and a constraint handling agreement in a single grammar specification to demonstrate our approach.

We start with the extraction constraints. [6] describes three different constraints on extraction, all of them expressed using the Constraint pattern (Figure 4(a)). We can take the union of these three G-ACGs to get the G-ACG on Figure 4(b). Next, we can incorporate knowledge from [8], by splitting the $\mathcal{L}_{Syn}$ lexicon and introducing a new intermediate signature (Figure 5(a)). The new signature, $Syntax$, deals purely with syntax without any issues of operator scope and its functions have lower-order types than those in Syn. We will finally

[9] This means that when the G-ACG is $\mathcal{D}$, the diamond grammar, and the node in question is $Bottom$, then we will have two different nodes for the two paths $[Top, Left, Bottom]$ and $[Top, Right, Bottom]$.



(a) Elaborating the lexicon from syntax to strings.

(b) Adding our own syntactic constraints, Neg and Agr.

Fig. 5: Building the example grammar up to its final state.

add our contribution, the signatures Neg and Agr and the lexicons $\mathcal{L}_{Neg}$ and $\mathcal{L}_{Agr}$ which implement the constraint on negation we discussed before and some simple notion of agreement, respectively (Figure 5(b)).

We notice that while the three constraints introduced in [6] are expressed in terms of the Syn signature, our constraints Agr and Neg use a different interface, the $Syntax$ signature, to constrain the languages being defined. The $Cont_i$ constraints are not actually constraining syntax itself, but the syntax-semantics interface. This is in some cases a deliberate decision, since e.g. $Cont_1$ is the constraint enforcing that quantifier scope must not escape outside of tensed clauses. The $Syntax$ signature is not equipped to talk about operator scope and so the constraint is expressed above the Syn signature. On the other hand, constraints $Cont_2$ and $Cont_3$ from [6] are purely syntactic and therefore could be more easily expressed in terms of $Syntax$ instead of Syn. However, we see it as a perk of our approach that we can ignore that and import the constraints wholesale without having to adapt them.

4.1 The *Neg* Signature

We will now look under the covers of the two signatures implementing the constraints, starting with negation and the Neg signature.

As was said in Section 2, we will proceed by splitting the atomic types N and NP into $N_{NEG=F}$ and $N_{NEG=T}$, and $NP_{NEG=F}$ and $NP_{NEG=T}$, respectively. We will use these new types to distinguish whether an N or NP phrase contains a negative determiner that can pair up with a negated verb. Where before we had a single type, we will now have possibly multiple types to account for the different values of the arguments' NEG features. Here are some representative examples:

$$
\begin{aligned}
N_{aucun} &: N_{NEG=F} \multimap NP_{NEG=T} \\
N_{le_1} &: N_{NEG=F} \multimap NP_{NEG=F} \\
N_{le_2} &: N_{NEG=T} \multimap NP_{NEG=T} \\
N_{ne_1^{tv}} &: (NP_{NEG=F} \multimap NP_{NEG=F} \multimap S) \multimap (NP_{NEG=T} \multimap NP_{NEG=F} \multimap S) \\
N_{ne_2^{tv}} &: (NP_{NEG=F} \multimap NP_{NEG=F} \multimap S) \multimap (NP_{NEG=F} \multimap NP_{NEG=T} \multimap S) \\
N_{ne_3^{tv}} &: (NP_{NEG=F} \multimap NP_{NEG=F} \multimap S) \multimap (NP_{NEG=T} \multimap NP_{NEG=T} \multimap S) \\
N_{aime} &: NP_{NEG=F} \multimap NP_{NEG=F} \multimap S \\
N_{que_1} &: (NP_{NEG=F} \multimap S) \multimap N_{NEG=F} \multimap N_{NEG=F} \\
N_{que_2} &: (NP_{NEG=F} \multimap S) \multimap N_{NEG=T} \multimap N_{NEG=T}
\end{aligned}
$$

We see that the negative determiner *aucun* can only attach to phrases which do not already contain a free negative determiner and that the resulting phrase is marked as having a free negative determiner. On the other hand, the determiner *le* does not have any interaction with the new NEG feature and is still able to take any N phrase, preserving any free negative determiners inside. The type of the negative particle *ne* modifies a verb (in the example above, $N_{ne_i^{tv}}$, a transitive verb) and produces a new verb that makes sure that at least one of its arguments contains a negative determiner. Verbs are not agnostic about the NEG feature. By default, they specifically demand it to be false, since a negative determiner cannot be coupled with a non-negated verb. Finally, the types of the relativizer *que* tell us two other things: the NP trace is considered as containing no negative determiners, and a relative clause does not care about or alter the presence of free negative determiners in the N it modifies.

The lexicon which translates from this signature to $Syntax$ is a straightforward relabeling. Its items can be characterized by the following schema:

$$\mathcal{L}_{Neg}(N_{wordform_i}) = C_{wordform}$$

where $C_{wordform}$ is the constant in $Syntax$ corresponding to the $wordform$.

4.2 The *Agr* Signature

The Agr signature will be constructed using the same strategy as Neg. N and NP will be subdivided into $N_{NUM=SG}$, $N_{NUM=PL}$, $NP_{NUM=SG}$ and $NP_{NUM=PL}$ (we only treat number agreement in this example). Here are some example types:

$$
\begin{aligned}
A_{tatou} &: N_{NUM=SG} \\
A_{tatous} &: N_{NUM=PL} \\
A_{le} &: N_{NUM=SG} \multimap NP_{NUM=SG} \\
A_{les} &: N_{NUM=PL} \multimap NP_{NUM=PL} \\
A_{aime_1} &: NP_{NUM=SG} \multimap NP_{NUM=SG} \multimap S \\
A_{aime_2} &: NP_{NUM=SG} \multimap NP_{NUM=PL} \multimap S
\end{aligned}
$$

$$A_{qui_1} : (NP_{NUM=SG} \multimap S) \multimap N_{NUM=SG} \multimap N_{NUM=SG}$$
$$A_{qui_2} : (NP_{NUM=PL} \multimap S) \multimap N_{NUM=PL} \multimap N_{NUM=PL}$$

The types for nouns and determiners are quite predictable. For the transitive verb *aime*, we see that we need a second type to account for the fact that *aime* does not care about the number of the object. Finally, the relativizer *qui* enforces that the number of the trace NP in the relative clause must be the same as the number of the N being modified.

The lexicon $\mathcal{L}_{Agr}$ follows exactly the same schema as that of $\mathcal{L}_{Neg}$.

4.3 Limits of Modularity

We have decomposed our grammar into 9 signatures and 8 lexicons. Besides making the grammar more readable and easily navigable, it also makes the grammar more attractive from a computational point of view. The complexity of parsing the highest abstract-most structures and verifying all the constraints is decomposed into smaller subtasks. A string can be parsed into its representation at the $Syntax$ level fairly easily (compared to parsing it at the higher levels) since $Syntax$ employs types of lower orders than the more abstract signatures. Finding the syntactic structure for a string is thus feasible. We then proceed to verifying the syntactic constraints. Here we can observe that the lexicons $\mathcal{L}_{Neg}$ and $\mathcal{L}_{Agr}$ are mere relabelings and therefore parsing (i.e. constraint verification) is trivial (i.e. decidable). Now in order to determine the meaning of our sentence, we have to continue parsing by finding an antecedent in Syn. This involves higher-order types used to encode the various scope-bearing operators. Finally, we can opt in to verify the high-level syntax-semantics constraints which employ dependent types. However, the lexicons in this case resemble relabelings and a semi-decidable procedure could be used to solve the problem. Having the grammar decomposed like this opens it up for computational processing by letting us engage in different tactics at different stages instead of having to parse a single hidden structure which enforces grammaticality w.r.t. every constraint. G-ACGs share these perks also with classical ACGs but they let use this composition in another dimension, which facilitates the writing of multiple constraints.

However, it is important to keep in mind that the composability demonstrated above is limited only to the composition of constraints. If we would like to extend our language with, e.g., a new lexical category, then every signature and every lexicon would have to be extended to cover this new category. It is also important to point out that the constraint signatures that we have been composing are not light-weight objects. They are usually defined by taking an existing signature that describes the language at a relevant level and by strengthening (usually all) the types of that signature to enforce the constraint[10]. However,

[10] This makes the expressivity of model-theoretic syntactic formalisms such as Interaction Grammars, in which adding the lexical items for *ne* and *aucun* along with enforcing the Neg constraint takes only two tree descriptions [10], even more striking when compared to our approach.

most of these type alterations are mundane and we believe that this is fertile ground for applying metagrammars.

5 Conclusion

We have considered the problem of building a wide-coverage ACG, specifically the problem of expressing a multitude of linguistic constraints. We have examined previous techniques and found no satisfying solution. We have thus provided an extension of the ACG formalism to solve the problem and justified the need for the increased expressivity. This embedding of syntactic constraints will contribute to an effort to define a syntax-semantics interface and later to build discourse structures.

In the end, our approach lets us define the syntax in a clean way using the idiomatic style of categorial grammars (simple atomic types like N, NP and S) and then define the constraints themselves the same way as they are defined in ACG research (such as is the case with [6]).

Interesting avenues of future work include: digging deeper into the language-theoretical properties of G-ACGs (Are ACG object languages closed on intersection? Are extrinsic languages as expressive as pangraphical languages?), searching for a metagrammatical description of constraint signatures to ease work on G-ACG grammars, and finally building large-scale grammars to verify the usability of our approach.

References

1. De Groote, P.: Towards abstract categorial grammars. In: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics (2001) 252–259
2. De Groote, P.: Towards a montagovian account of dynamics. In: Proceedings of SALT. Volume 16. (2006)
3. Qian, S., Amblard, M.: Event in compositional dynamic semantics. In: Logical Aspects of Computational Linguistics. (2011)
4. Asher, N., Pogodalla, S.: Sdrt and continuation semantics. In: New Frontiers in Artificial Intelligence. Volume 6797 of Lecture Notes in Computer Science. (2011)
5. Asher, N., Pogodalla, S.: A montagovian treatment of modal subordination. In: Proceedings of SALT. Volume 20. (2011) 387–405
6. Pogodalla, S., Pompigne, F.: Controlling extraction in abstract categorial grammars. In: Formal Grammar. (2012)
7. de Groote, P.: Tree-adjoining grammars as abstract categorial grammars. In: TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks, Università di Venezia (2002) 145–150
8. Pogodalla, S.: Generalizing a proof-theoretic account of scope ambiguity. In: 7th International Workshop on Computational Semantics. (2007)
9. Montague, R.: The proper treatment of quantification in ordinary english
10. Perrier, G.: A french interaction grammar. In: International Conference on Recent Advances in Natural Language Processing. (2007)
11. de Groote, P., Maarek, S.: Type-theoretic extensions of abstract categorial grammars. ESSLLI 2007 (2007)